DAM
Desarrollo de Aplicaciones Multiplataforma
2º Curso

# AD
# Acceso a Datos

UD 3
Manejo de Conectores
(JDBC)

IES BALMIS
Dpto Informática
Curso 2019-2020
Versión 1 (03/2019)

# UD3 – Manejo de conectores

## ÍNDICE

# 5. Conexión JDBC a MySQL

Supongamos que tenemos una BD en MySQL con el nombre **biblioteca** y en ella la siguiente tabla libros con 3 registros:

```
CREATE SCHEMA biblioteca DEFAULT CHARACTER SET utf8mb4 ;

USE biblioteca ;

CREATE TABLE libros (
  id        int(11) NOT NULL,
  titulo    varchar(60) DEFAULT NULL,
  autor     varchar(60) DEFAULT NULL,
  PRIMARY KEY (id)
);

INSERT INTO libros VALUES
      (1,'Macbeth','William Shakespeare'),
      (2,'La Celestina (Tragicomedia de Calisto y Melibea)',
            'Fernando de Rojas'),
      (3,'El Lazarillo de Tormes','Anónimo');
```

A nivel genérico, los pasos necesarios serán:

- Abrir una conexión.
- Preparar un objeto de tipo "**Statement**", que representará a una (o varias) sentencia(s) SQL.
- A través de ese "Statement", lanzar consultas y obtener resultados.
- Usar excepciones para gestionar los posibles errores.

Ya como aplicación real, los pasos que emplearemos para conectar y obtener un bloque de información son los siguientes:

- Usar "**Class.forName**" para cargar el driver que se haya escogido. En nuestro caso será:

```
Class.forName("com.mysql.jdbc.Driver");
```

- Establecer una conexión, indicando la URL del servidor, el usuario que conecta y la contraseña. Será algo como:

```
Connection con = DriverManager.getConnection(url, usuario, password);
```

- Preparar una sentencia que se ejecutará sobre esa conexión:

```
Statement statement = con.createStatement();
```

- La orden SQL concreta a ejecutar se indicará en un método "**executeQuery**". Si se trata de una consulta que devuelva datos, éstos estarán accesibles a través de un "conjunto de resultados" ("**ResultSet**"):

```
ResultSet rs = statement.executeQuery("SELECT id, titulo, autor FROM libros");
```

- Ese conjunto de resultados se recorrerá de forma secuencial: "**rs.next()**" será verdadero si hay más datos en ese conjunto de resultados:

```
while (rs.next()) {
  ...
}
```

Si cada fila del resultado está formada por varios campos, podemos obtener el valor de cada uno de ellos con "**rs.getString( x )**", donde x puede ser un número (la posición de esa columna en la consulta, comenzando con 1) o un nombre de campo (indiferente de mayúsculas o minúsculas):

```
String titulo  = rs.getString("titulo");
String autor   = rs.getString(3);
```

(si un campo es numérico, no hace falta leerlo como cadena y posteriormente convertirlo a número, sino que se puede emplear otros getters como "**.getInt(1)**".

- Finalmente, se deberá cerrar el ResultSet y la conexión:

```
rs.close();
con.close();
```

En nuestros primeros ejemplos no haremos ningún control de errores, nos limitaremos a permitir que "**main**" lance la correspondiente excepción (lo que provocaría el final de la ejecución).

Así, un ejemplo completo podría ser:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class ConsultaBD1 {
  public static void main(String[] args)
                    throws ClassNotFoundException, SQLException {
    // Conexión a la BD
    String url;
    Class.forName("com.mysql.jdbc.Driver");
    url = "jdbc:mysql://localhost:3306/biblioteca";
    url +="?autoReconnect=true&useSSL=false&zeroDateTimeBehavior=convertToNull";
    String usuario = "root";
    String password = "1234";
    Connection con = DriverManager.getConnection(url, usuario, password);

    // Crear Statement de la Consulta
    String sentenciaSQL = "SELECT titulo, autor FROM libros";
    Statement statement = con.createStatement();

    // Resulset
    ResultSet rs = statement.executeQuery(sentenciaSQL);
    System.out.printf("%-60s %-60s\n", "Título", "Autor");
    System.out.print("------------------------------------------------------------");
    System.out.print(" ");
    System.out.print("------------------------------------------------------------");
    System.out.println();
    while (rs.next()) {
      System.out.printf("%-60s %-60s\n", rs.getString("titulo"), rs.getString("autor"));
    }
    rs.close();

    // Cerrar conexión
    con.close();
  }
}
```

## ***Errores***

(En un caso real, es habitual que en vez de "localhost" se indique la dirección IP del servidor, que no tiene por qué coincidir con el cliente, por ejemplo 192.168.0.100).

Es posible que al lanzar este programa obtengamos el mensaje de error:

```
Exception in thread "main" java.lang.ClassNotFoundException:
org.mysql.jdbc.Driver at ...
```

porque ese fichero (el driver JDBC de MySQL) no se encuentre en el “class path” (la ruta de búsqueda para clases auxiliares del programa).

## ***Compilar y lanzar aplicación desde consola***

Si se compila o se lanza desde el terminal, es posible indicar la ruta del driver JDBC usando la opción "-cp" (o "-classpath"):

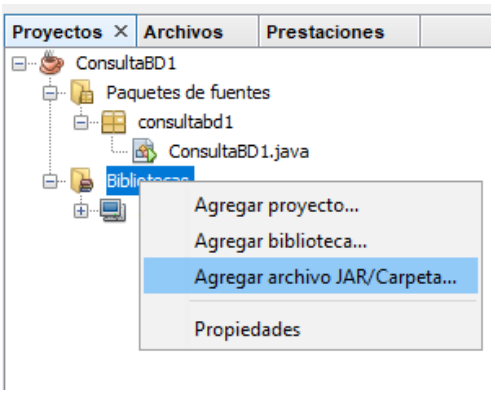
```
javac -cp mysql-connector-java-5.1.48-bin.jar; ConsultaBD1.java
java  -cp mysql-connector-java-5.1.48-bin.jar; ConsultaBD1
```

## ***Netbeans***

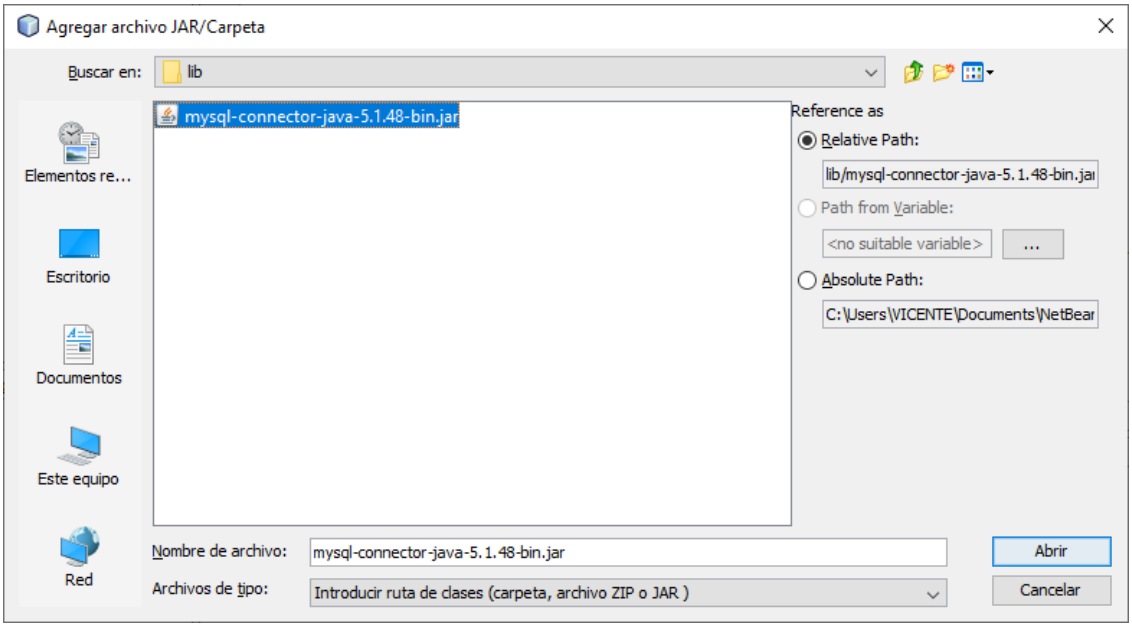
Desde NetBeans también se podrá crear con facilidad proyectos que accedan a bases de datos.

La forma de añadir una biblioteca a un proyecto es pulsar el botón derecho sobre el proyecto para acceder a sus "**Bibliotecas**" del proyecto.

Y ahí tendremos una opción "**Agregar archivo JAR/Carpeta**".




Crearemos una carpeta denominada **lib** en nuestro proyecto y la añadiremos con path relativo, para que el proyecto sea portable.




Todos los archivos **jar** incluidos en esta carpeta se añadirán al proyecto. En nuestro ejemplo, añadiremos "**mysql-connector-java-5.1.48-bin.jar**".

Una vez esa biblioteca esté añadida al proyecto, éste ya debería compilar y funcionar correctamente.

# 6. Sentencias de definición de datos

Hemos visto que se pueden lanzar consultas que devuelvan datos con "**.executeQuery**". Si se trata de consultas de definición de datos (parte del DDL, Data Definition Language), como CREATE, ALTER o DROP, se deberá emplear "**.executeUpdate**". Ejemplo:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class ConsultaBD2 {
  public static void main(String[] args)
                    throws ClassNotFoundException, SQLException {

    // Conexión a la BD
    String url;

    Class.forName("com.mysql.jdbc.Driver");
    url = "jdbc:mysql://localhost:3306/biblioteca";
    url +="?autoReconnect=true&useSSL=false&zeroDateTimeBehavior=convertToNull";
    String usuario = "root";
    String password = "1234";
    Connection con = DriverManager.getConnection(url, usuario, password);

    // Crear Statement del CREATE TABLE
    String sentenciaSQL = "CREATE TABLE personas ( "+
                "codigo VARCHAR(4) PRIMARY KEY, "+
                "nombre VARCHAR(50), "+
                "email VARCHAR(40) "+ ");";
    Statement statement = con.createStatement();

    // Execute
    statement.executeUpdate(sentenciaSQL);
    System.out.println("Tabla personas creada");

    // Cerrar conexión
    con.close();
  }
}
```

# 7. Sentencias de manipulación de datos

Al igual que para las sentencias SQL de definición de datos, las de manipulación de datos (parte del DML, como INSERT, UPDATE o DELETE) no devolverán un conjunto de datos, y se deberán llamar usando "**.executeUpdate**". Eso sí, en este caso puede merecer la pena comprobar el valor devuelto, que es un número entero, que representa la cantidad de filas afectadas (y que en el caso de las sentencias de definición de datos se podía ignorar, porque siempre vale cero).

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class ConsultaBD3 {
  public static void main(String[] args)
                    throws ClassNotFoundException, SQLException {

    // Conexión a la BD
    String url;

    Class.forName("com.mysql.jdbc.Driver");
    url = "jdbc:mysql://localhost:3306/biblioteca";
    url +="?autoReconnect=true&useSSL=false&zeroDateTimeBehavior=convertToNull";
    String usuario = "root";
    String password = "1234";
    Connection con = DriverManager.getConnection(url, usuario, password);

    // Crear Statement del Insert
    String sentenciaSQL = "INSERT INTO libros (id, titulo, autor) VALUES "+
               "(4,'20.000 Leguas de Viaje Submarino', 'Julio Verne'), "+
               "(5,'Alicia en el País de las Maravillas','Lewis Carrol');";
    Statement statement = con.createStatement();

    // Execute
    int cantidad = statement.executeUpdate(sentenciaSQL);
    System.out.println("Datos insertados: " + cantidad);


    // Cerrar conexión
    con.close();
  }
}
```

La salida de este programa debería ser:

```
Datos insertados: 2
```

Y si ahora volvemos a lanzar el programa ConsultaBD1, que mostraba todos los datos, deberían mostrarse 5.

```
Titulo                                                Autor
----------------------------------------------------- ------------------------------
Macbeth                                               William Shakespeare
La Celestina (Tragicomedia de Calisto y Melibea)      Fernando de Rojas
El Lazarillo de Tormes                                Anónimo
20.000 Leguas de Viaje Submarino                      Julio Verne
Alicia en el País de las Maravillas                   Lewis Carrol
```

La otra senencia que forma parte del DML es SELECT, que, como ya hemos visto anteriormente, sí devuelve en general un conjunto de datos, y se lanza con "**.executeQuery**".

Otra opción a la hora de realizar consultas es la utilización de Sentencias preparadas para luego indicar el valor de los parámetros. Para esto utilizaremos los objetos "**PreparedStatement**".

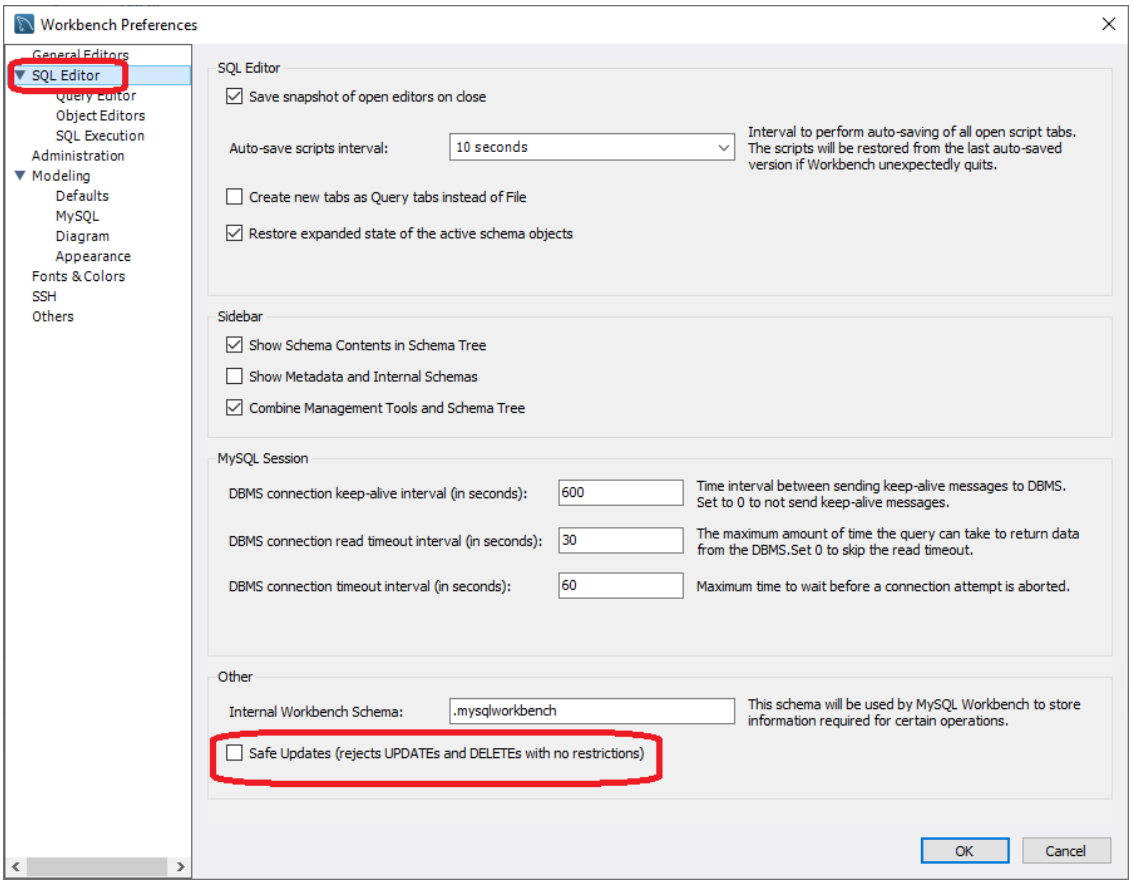
```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.PreparedStatement;

public class ConsultaBD4 {
  public static void main(String[] args)
                    throws ClassNotFoundException, SQLException {
    // Conexión a la BD
    String url;

    Class.forName("com.mysql.jdbc.Driver");
    url = "jdbc:mysql://localhost:3306/biblioteca";
    url +="?autoReconnect=true&useSSL=false&zeroDateTimeBehavior=convertToNull";
    String usuario = "root";
    String password = "1234";
    Connection con = DriverManager.getConnection(url, usuario, password);

    // Crear Statement de la Consulta
    String sentenciaSQL = "SELECT titulo, autor FROM libros WHERE autor = ?";
    PreparedStatement pStatement = con.prepareStatement(sentenciaSQL);
    pStatement.setString(1, "Julio Verne");

    // Resulset
    ResultSet rs = pStatement.executeQuery();
    System.out.printf("%-60s %-60s\n", "Título", "Autor");
    System.out.print("------------------------------------------------------------");
    System.out.print(" ");
    System.out.print("------------------------------------------------------------");
    System.out.println();
    while (rs.next()) {
      System.out.printf("%-60s %-60s\n", rs.getString("titulo"), rs.getString("autor"));
    }
    rs.close();

    // Cerrar conexión
    con.close();
  }
}
```

# 8. Ejecución de procedimientos

En varias BD como Oracle, MySQL y PostgreSQL se permite definir funciones usando su lenguaje de programación propio como el PL/SQL.

En MySQL, para poder crear procedimientos y funciones sin restricciones, deberemos realizar dos cambios en la configuración:

1) En Workbench, accederemos a "Edit → Preferences" y vamos a desactivar el control seguro de actualizaciones de datos:

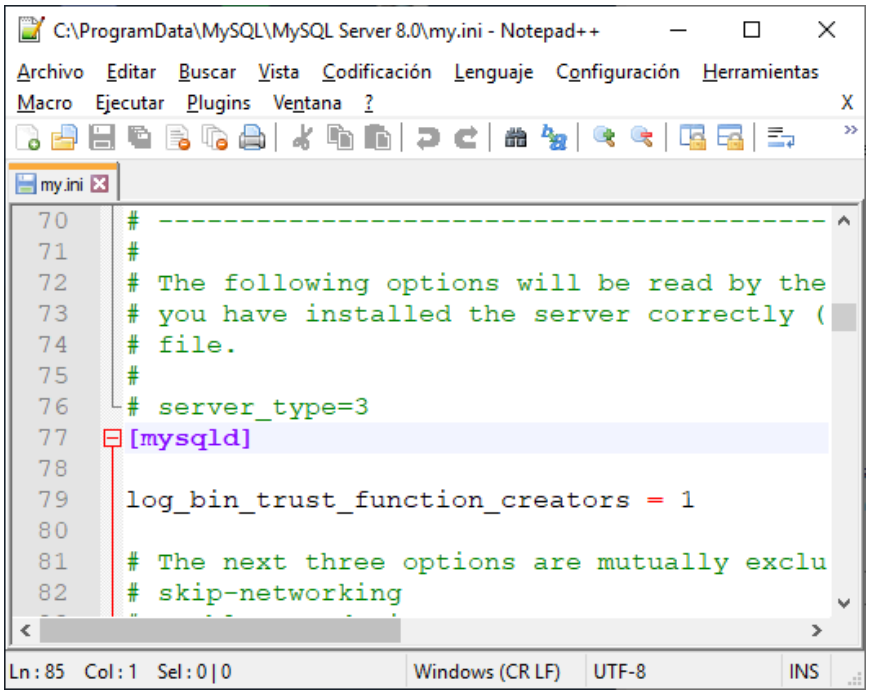


2) Cambiar la configuración para poder crear funciones sin restricciones de accesoa datos. Para ello editaremos el archivo de configuración:

**`C:\ProgramData\MySQL\MySQL Server 8.0\my.ini`**

Añadiremos en la sección [mysqld] la siguiente línea:

```
log_bin_trust_function_creators = 1
```




Después de estos dos cambios, deberemos reiniciar el servicio de **MySQL80** desde **services.msc**

Un ejemplo sencillo en MySQL de un procedimiento es:

```
/* Eliminar el procedimiento si ya existe */
DROP PROCEDURE IF EXISTS maximo;

/* Crear el procedimiento */
DELIMITER //
CREATE PROCEDURE maximo(IN num1 DECIMAL(11,2),
                        IN num2 DECIMAL(11,2),
                        OUT num3 DECIMAL(11,2))
  BEGIN
      IF (num1>num2) THEN
            SET num3 := num1;
      ELSE
            SET num3 := num2;
      END IF;

  END //
DELIMITER ;
```

A este tipo de funciones se les suele llamar también "**procedimientos almacenados**" (**Stored Procedures**) y se pueden llamar luego desde otros lenguajes en el equipo cliente. De hecho, es una de las formas de trabajar que se suelen recomendar para evitar ataques de "inyección de SQL".

```
CALL maximo(10.3, 21.4, @num);
SELECT @num;
```

De igual manera, las funciones (procedimientos almacenados) pueden acceder a datos de la BD.

En Java es posible crear un programa que llame a esa función, que le pase parámetros y que obtenga sus resultados.

Para ello, emplearemos un "**CallableStatement**":

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.CallableStatement;
import java.sql.Types;

public class EjemploDeStoredProcedure {
  public static void main(String[] args)
                    throws ClassNotFoundException, SQLException {

    // Conexión a la BD
    String url;

    Class.forName("com.mysql.jdbc.Driver");
    url = "jdbc:mysql://localhost:3306/biblioteca";
    url +="?autoReconnect=true&useSSL=false&zeroDateTimeBehavior=convertToNull";
    String usuario = "root";
    String password = "1234";
    Connection con = DriverManager.getConnection(url, usuario, password);

    // Execute
    CallableStatement cStmt = con.prepareCall("{call maximo(?,?,?)}");
    cStmt.setFloat(1, (float) 10.4);
    cStmt.setFloat(2, (float) 30.7);
    cStmt.registerOutParameter(3, Types.DECIMAL);

    cStmt.execute();
    float resultado = cStmt.getFloat(3);
    System.out.println("El máximo es: " + resultado);

    // Cerrar conexión
    con.close();
  }
}
```

Si la función es más sencilla, es decir, que no tiene parámetros, quizá no se pueda llamar con un "**callable statement**", porque obtengamos un error diciendo que intentamos acceder a la columna 1 de 0 columnas disponibles en "**registerOut-Parameter**".

Esto no es grave, porque no se trataría de algo susceptible de ataques como los de inyección de SQL. Deberíamos realizar una función en vez de un procedimiento y se podría acceder simplemente con un SELECT.

Si creamos una función que devuelva el número de libros existentes:

```
/* Eliminar la función si ya existe */
DROP FUNCTION IF EXISTS totallibros;

/* Crear la función */
DELIMITER //
CREATE FUNCTION totallibros()
      RETURNS INT
  BEGIN
      DECLARE total INT;

      SELECT COUNT(*) INTO total FROM libros;

      RETURN total;
  END //
DELIMITER ;
```

Para llamarla basta con utilizar SELECT.

```
SELECT totallibros();
```

Ahora para llamarla utilizaremos un statement normal, como los de SELECT a la BD.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;

public class LlamadaAFuncionSencilla {
  public static void main(String[] args)
                    throws ClassNotFoundException, SQLException {

    // Conexión a la BD
    String url;

    Class.forName("com.mysql.jdbc.Driver");
    url = "jdbc:mysql://localhost:3306/biblioteca";
    url +="?autoReconnect=true&useSSL=false&zeroDateTimeBehavior=convertToNull";
    String usuario = "root";
    String password = "1234";
    Connection con = DriverManager.getConnection(url, usuario, password);

    // Execute
    Statement statement = con.createStatement();
    String consulta = "SELECT totallibros()";

    ResultSet rs = statement.executeQuery(consulta);
    rs.next();
    System.out.println("El total de libros es: " + rs.getInt(1));

    // Cerrar conexión
    con.close();
  }
}
```

# 9. Gestión de errores

Hasta ahora, no estamos tratando los errores de ninguna forma, simplemente estamos dejando que "**main**" se interrumpa lanzando la correspondiente excepción;

```
  public static void main(String[] args)
                    throws ClassNotFoundException, SQLException {
```

Una alternativa un poco más amigable es usar un bloque **try-catch** para atrapar el error, informar al usuario y permitir que el programa prosiga a pesar del problema.

Una primera aproximación será simplemente mostrar el error existente usando "**printStackTrace()**". Por ejemplo, podemos crear una versión alternativa del programa **ConsultaBD2**, que creaba una tabla, pero añadiendo esta vez la posibilidad de que avise en caso de que no se haya podido crear la tabla (cosa que ocurrirá en esta segunda ocasión, en que la tabla ya existe):

```
import java.sql.*;

public class ConsultaBD1gestionerrores {
 public static void main(String[] args) {
  try {
    // Conexión a la BD
    String url;

    Class.forName("com.mysql.jdbc.Driver");
    url = "jdbc:mysql://localhost:3306/biblioteca";
    url +="?autoReconnect=true&useSSL=false&zeroDateTimeBehavior=convertToNull";
    String usuario = "root";
    String password = "1234";
    Connection con = DriverManager.getConnection(url, usuario, password);

    // Crear Statement de la Consulta
    String sentenciaSQL = "SELECT titulo, autor FROM libros";
    Statement statement = con.createStatement();

    // Resulset
    ResultSet rs = statement.executeQuery(sentenciaSQL);
    System.out.printf("%-60s %-60s\n", "Título", "Autor");
    System.out.print("------------------------------------------------------------");
    System.out.print(" ");
    System.out.print("------------------------------------------------------------");
    System.out.println();
    while (rs.next()) {
      System.out.printf("%-60s %-60s\n", rs.getString("titulo"), rs.getString("autor"));
    }
    rs.close();

    // Cerrar conexión
    con.close();
  } catch (ClassNotFoundException ce) {
    ce.printStackTrace();
  } catch (SQLException se) {
    se.printStackTrace();
  }
  System.out.println("Terminado!");
 }
}
```

Como se puede ver, aparece el mensaje "Terminado!" al final, lo que indica que la ejecución no se ha interrumpido aunque se muestre algún error.

Aun así, en general, será preferible analizar los posibles errores por separado, de modo que los mensajes de error (y la posible recuperación de éstos) sean más específicos:

```
...

      // Cerrar conexión
      con.close();
    } catch (ClassNotFoundException ce) {
      // ce.printStackTrace();
      System.out.println("MySQL no accesible");
    } catch (SQLException se) {
      // se.printStackTrace();
      System.out.println("No se ha podido realizar el SELECT");
    }
    System.out.println("Terminado!");
  }
}
```

Lo que mostraría un mensaje más razonable.

De hecho, incluso una "**SQLException**" es algo demasiado genérico, de modo que en un caso real se podrían analizar los valores de "**getMessage()**" y de "**getErrorCode()**" para saber el tipo de error concreto que ha existido (en general, esto puede ser preferible a usar varios bloques try-catch en un fragmento de código tan pequeño, para separar la lógica de programa de la lógica de la gestión de errores, que es lo que pretende el uso de try-catch.

Otra opción de mejora, sería crear un **finally** para asegurar que cerramos objetos.

```
    ...
    Connection con = null;
    Statement statement = null;
    try {
      ...
      con = DriverManager.getConnection(url, usuario, password);
      ...
      statement = con.createStatement();
      ...
    } catch (ClassNotFoundException ce) {
      // ce.printStackTrace();
      System.out.println("MySQL no accesible");
      System.out.println(ce.getMessage());
    } catch (SQLException se) {
      // se.printStackTrace();
      System.out.println("No se ha podido realizar el SELECT");
      System.out.println(se.getErrorCode()+" "+se.getMessage());
    } finally {
      if ( statement != null) {
        try { statement.close(); } catch (SQLException ex) { } ;
      }
      if (con != null) {
        try { con.close(); } catch (SQLException ex) { } ;
      }
    }
    System.out.println("Terminado!");
  }
}
```